SynFuzz: Leveraging Fuzzing of Netlist to Detect Synthesis Bugs

Raghul Saravanan
George Mason University
Fairfax, Virginia, USA
rsaravan@gmu.edu

Sudipta Paria
University of Florida
Gainesville, Florida, USA
sudiptaparia@ufl.edu

Aritra Dasgupta
University of Florida
Gainesville, Florida, USA
aritradasgupta@ufl.edu

Venkat Nitin Patnala
George Mason University
Fairfax, Virginia, USA
vpatnala@gmu.edu

Swarup Bhunia
University of Florida
Gainesville, Florida, USA
swarup@ece.ufl.edu

Sai Manoj P D
George Mason University
Fairfax, Virginia, USA
spudukot@gmu.edu

Abstract

In the evolving landscape of integrated circuit (IC) design, the increasing complexity of modern processors and intellectual property (IP) cores has introduced new challenges in ensuring design correctness and security. Recent attacks targeting hardware vulnerabilities have heightened the need for enhanced security measures throughout the design flow. The recent advancements in hardware fuzzing techniques have shown their efficacy in detecting hardware bugs and vulnerabilities at the RTL abstraction level of hardware. However, they suffer from several limitations, including an inability to address vulnerabilities introduced during synthesis and gate-level transformations. These methods often fail to detect issues arising from library adversaries, where compromised or malicious library components can introduce backdoors or unintended behaviors into the design. This gap leaves critical flaws undetected and underscores the need for more comprehensive fuzzing techniques that extend beyond the RTL level to ensure hardware integrity and security. In this paper, we present a novel hardware fuzzer, SynFuzz, designed to overcome the limitations of existing hardware fuzzing frameworks. SynFuzz focuses on fuzzing hardware at the gate-level netlist to identify synthesis bugs and vulnerabilities that arise during the transition from RTL to the gate-level. We analyze the intrinsic hardware behaviors using coverage metrics specifically tailored for the gate-level. Furthermore, SynFuzz implements differential fuzzing to uncover bugs associated with EDA libraries. We evaluated SynFuzz on popular open-source processors and IP designs, successfully identifying 7 new synthesis bugs. Additionally, by exploiting the optimization settings of EDA tools, we performed a compromised library mapping attack (CLiMA), creating a malicious version of hardware designs that remains undetectable by traditional verification methods. We also demonstrate how SynFuzz overcomes the limitations of the industry-standard formal verification tool, Cadence Conformal, providing a more robust and comprehensive approach to hardware verification.

1 Introduction

The intricate global semiconductor supply chain demands seamless collaboration between IC designers and vendors, with various entities playing vital roles at every stage of the IC life cycle, including design, verification, fabrication, and integration [9, 24]. For example, the design of the Apple® A15 chip involved 11 third-party entities to deliver sophisticated solutions [89]. However, this intricate design process is prone to trust issues, such as bugs and vulnerabilities,

due to opaque interactions between entities[8, 55, 60, 61, 75]. The growing dependence on third-party intellectual property (3P-IP) blocks exponentially expands the scope for exploiting design vulnerabilities. These vulnerabilities stem from the interaction of hardware and software components in modern IC designs, especially system-on-chips (SoCs) and microprocessors [71, 72]. The number of identified common vulnerability enumerations (CVEs) recorded in 2024 is close to 29,004, increased by 43% compared to 2021 [29]. Moreover, bugs emerging from the hardware are irreversible and pose a significant challenge to the integrity of the system.

The hardware vulnerabilities can manifest at various levels of abstraction within the design flow, including the design phase (Register Transfer Level (RTL)), the synthesis stage (gate-level netlist), and the implementation phase (physical layout) [3, 15, 35, 38, 51, 54, 66, 81, 88, 93, 97]. The sophisticated IC design flow relies on state-of-the-art Electronic Design Automation (EDA) tools [4, 16–18, 70, 83, 98] that facilitate seamless transitions across different hardware abstraction levels. However, the inherent nature of the hardware design flow exposes EDA tools to potential threats from adversarial entities, including malicious developers or compromised vendors. These entities, often treated as trusted, can manipulate the EDA tools to inject subtle but significant bugs, resulting in a malicious piece of hardware design [90]. Thus, hardware verification is essential at each abstraction in the design flow to meet the functional and security requirements [73, 90].

During the synthesis stage, the IP designs, expressed in Hardware Description Languages (HDL) at the RTL, are translated to an equivalent gate-level netlist representation through EDA logic synthesis tools. This process involves a variety of logic optimizations to meet performance, power, and area (PPA) requirements by mapping the HDL design to instances of standard-cell gates provided by the library vendors as binary libraries. These libraries, which are tailored to a specific technology node, represent the functional and physical properties of the gates and logic cells used in the design. The accuracy of this translation and mapping is crucial, as any inconsistencies can lead to functional errors or vulnerabilities in the final implementation phase.

To ensure the translated design adheres to the intended behavior specified at the RTL level, formal verification methods such as Logic Equivalence Check (LEC), often leveraging techniques such as Bounded Model Checking (BMC) [10, 11], are employed to compare the gate-level netlist with the RTL design and identify any discrepancies or unintended changes [41]. However, these formal verification methods face significant scalability challenges and

heavily rely on human expertise, limiting their efficacy in bug detection for complex designs [7, 58, 59, 62, 72, 91, 95, 100]. Hence, there is a compelling need for methodologies and tools to capture the bugs and vulnerabilities introduced during the synthesis process.

Hardware fuzzing [49, 52, 53, 56, 57, 67, 92], inspired by software fuzzing [39, 40, 63], has gained significant traction due to its effectiveness in detecting bugs in complex hardware designs. To date, numerous proposals have focused on developing hardware fuzzing methodologies aimed at identifying bugs at the RTL level, particularly in the designs of CPU architectures [21, 53, 56, 80]. However, these methodologies are inherently limited by the chosen level of abstraction, which focuses exclusively on the pre-synthesis stage. This abstraction overlooks the transformations and optimizations that occur during synthesis, where subtle yet impactful bugs can be introduced. As a result, existing fuzzing frameworks are inadequate for detecting synthesis-induced bugs that manifest at the gate-level netlist stage. This gap underscores the necessity for innovative fuzzing approaches that can operate effectively at the gate-level netlist to verify the functional correctness between the RTL and the gate-level netlist.

Our Proposed Work: In this work, we propose **SynFuzz**, a novel hardware fuzzer that leverages fuzzing at the gate-level netlist aimed at detecting synthesis and library-related bugs. To the best of our knowledge, this is the very first work on hardware fuzzing at gate-level netlist. The input to our **SynFuzz** is a gate-level abstraction of the design, where the HDL is synthesized and mapped to library cell gates provided by the standard-cell library. **SynFuzz** captures intrinsic hardware behaviors associated with library cell gates, enabling the detection of subtle inconsistencies introduced during the synthesis process.

Furthermore, we show that a maliciously modified library, designed to appear more attractive to EDA tools during the optimization and synthesis stages, can be exploited to introduce a novel threat model, termed **CLiMA (Compromised Library Mapping Attack)**. CLiMA enables flawed library mapping during the synthesis process, enabling targeted attacks on the designs, and resulting in unintended hardware implementation. We leverage CLiMA to design a malicious version of the or1200 CPU design, evading a leading formal verification tool, Cadence® Conformal. In contrast to software fuzzers [1, 39, 40] that rely on crashes, and hardware fuzzers that depend on golden models [27, 56] or ISA for validation [21, 53], EDA libraries lack any equivalent golden models. In this work, we address this challenge by identifying library bugs in the context of differential fuzzing. As such, **SynFuzz** *1) supports conventional hardware and design verification flow 2) detects synthesis and library vulnerabilities during the synthesis stage 3) does not require extensive design or EDA tool knowledge 4) is scalable to large and complex designs.* In summary, our key contributions are :

- We propose our new hardware fuzzer, **SynFuzz** that leverages state-of-the-art gate-level fuzzing at post-synthesis level to unveil RTL, synthesis and library bugs(Section 4).
- This work introduces a new class of synthesis attack model CLiMA (Section 6), leveraging malicious library mapping through the logic synthesis tools.

- We leverage CLiMA to inject bugs in the or1200 CPU design that evades formal verification methods such as Logic Equivalence Check (Section 6.1, 6.2).
- We extensively evaluate our fuzzer, **SynFuzz**, on four popular and complex open-source CPU designs: 1) the or1200 processor (OpenRISC ISA) [42], 2) the IBEX processor (RISC-V ISA) [43], 3) the PicoRV32 processor (RISC-V ISA) [43], and 4) the MIPS processor [64]. Additionally, we test it on cryptographic cores such as RSA from TrustHub [79] and Data Encryption Standard (DES) [65], a Digital Signal Processor (DSP) [50], and several IP peripherals, including UART [50], GPIO [50].
- **SynFuzz** found 7 new synthesis bugs across three processors and one cryptographic core, with all 7 being newly discovered bugs (Section 5.2). In addition, **SynFuzz** found 10 existing synthesis translation bugs in popular open-source EDA synthesis tool Yosys. To evaluate the efficacy of our fuzzer, we utilized a leading formal verification tool, Cadence's Conformal Logic Equivalence Check [20]. **SynFuzz** addresses the limitations of the Conformal tool, including scalability issues and its heavy reliance on human expertise (Section 5.5).

2 Background

In this section, we furnish the necessary background on hardware fuzzing, and the hardware development cycle.

2.1 From Design to Threats: Unpacking the Modern IC Lifecycle

The recent advancements in technology and the rising demand for enhanced performance, energy efficiency, and robust security have significantly increased the complexity of the modern IC design process. However, throughout the IC development lifecycle, various threats can emerge at different stages, from design to deployment, as shown in Figure 1. Starting with defining system-level requirements and specifications in the design stage, the functional, performance, and security objectives are outlined for the IC. IP vendors generate the register-transfer level (RTL) code using hardware description languages such as Verilog, VHDL, and SystemVerilog, which align with the design descriptions through data flow between registers and the logic operations that occur during each clock cycle. The RTL design undergoes thorough verification to ensure it meets the functional requirements [6, 12, 36, 78]. This verification is typically carried out using RTL simulations via EDA tools like Cadence Xcelium, Synopsys VCS [16, 17, 86, 87] which apply test vectors to check for correctness and identify any functional bugs in the design. During this stage, risks such as inaccurate requirements, design flaws, and Trojan insertion can result in IP theft, data leakage, and backdoor exploitations and several countermeasures are proposed to protect against diverse attacks [3, 15, 22, 28, 35, 38, 51, 54, 66, 74, 76, 77, 81, 88, 93, 97].

In the next phase, EDA tools perform synthesis that translates the RTL design into a gate-level representation. This step maps the design onto fundamental hardware components such as logic gates, flip-flops, buffers, and other standard cells from the technology library provided by the semiconductor foundry. However, it is essential to ensure that the gate-level design meets both functional
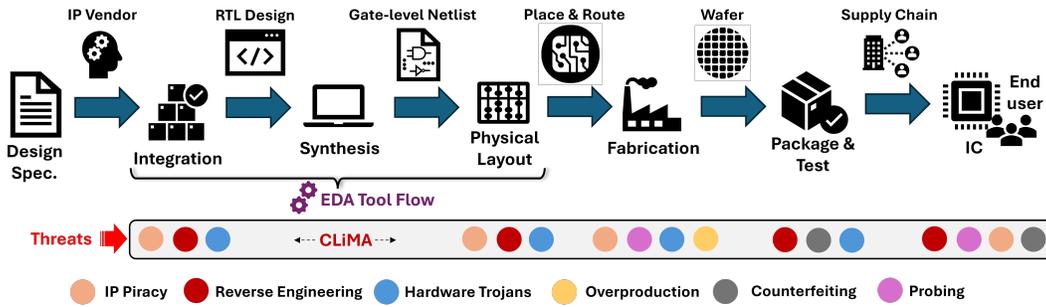
Figure 1: Threat Space in the Modern IC Design Flow

and timing requirements [34] requiring through verification efforts. The current state-of-the-art techniques, including formal methods, functional simulations, and fault simulations, are employed to compare the design against the RTL specifications [16, 18, 78]. During synthesis and post-synthesis verification, issues such as timing violations, incomplete verification, and malicious modifications can arise [84]. The adversary can exploit the EDA tool optimizations during synthesis to introduce bugs that propagate to later stages, resulting in IP piracy, denial-of-service, confidentiality violations, etc [32]. *Based on these threat spaces identified during the synthesis stage, this serves as the primary motivation to extend fuzzing methodologies to the gate-level netlist.* By targeting this critical abstraction level, it becomes possible to identify synthesis-induced bugs and vulnerabilities that are otherwise missed by traditional RTL-level fuzzing approaches.

The design then moves into the physical design phase, where the gate-level netlist is converted into a layout, followed by physical verification, preparing the design for tape-out and subsequent manufacturing at the foundry. At this stage, all verification occurs in a pre-silicon environment, meaning the design is represented as RTL, gate-level, or layout models, not yet as physical silicon. A key advantage of pre-silicon verification is the high degree of observability, which allows engineers to monitor and debug the design thoroughly. This level of visibility is not possible in post-silicon validation. The physical design and fabrication stages are vulnerable to threats such as overproduction, counterfeit ICs, hardware Trojans, and probing attacks, all of which can significantly undermine the chip's reliability and trustworthiness [13].

Once the chip is fabricated, post-silicon verification is employed which is more complex and costly than pre-silicon verification, offering limited observability. Its primary goal is to confirm that the manufactured silicon meets the design specifications and functions as intended. Throughout both the pre-and post-silicon verification stages, each level of the design is compared to its previous respective Golden Reference Model (GRM) to ensure accuracy and reliability.

2.2 Fuzzing

Fuzzing involves bombarding randomized input seeds to the Program Under Test (PUT)/Design Under Test (DUT), intending to trigger the existing bugs and vulnerabilities. The generated input seeds are further mutated based on the coverage feedback obtained from the DUT/PUT under investigation, as shown in Figure 2. The popular mutation algorithms used by fuzzers are American Fuzzy Lop (AFL) mutation algorithms [39] such as bit-flip, swap, arithmetic, etc. The coverage feedback aids in steering the fuzzer to generate effective seeds for improved coverage and discard uninteresting inputs. Based on the various mechanisms to obtain coverage feedback, fuzzers are classified as white-box, grey-box, and black-box fuzzing. The outcomes from the DUT/PUT are analyzed for any crash or GRM implementation to detect the bugs. Fuzzing has proven its efficiency in finding vulnerabilities in software [1, 37, 40, 63] and hardware platforms [16, 26, 30, 31, 33, 41, 44–47, 96]. Industry-based fuzzing frameworks such as Google's OSS-Fuzz [82] and Microsoft's Security Risk Detection [63] have proved their efficacy and effectively identified a plethora of security vulnerabilities. The popular software fuzzer, AFL [39], is predominantly used in software fuzzing and is predominantly used in the majority of the software fuzzing techniques [1, 37, 40, 63].
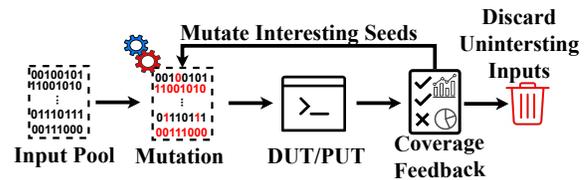

Figure 2: Overview of Fuzzing

In the context of hardware fuzzing, the fuzzing target is at different levels of abstraction including RTL or gate-level netlist or physical layout. There are three ways to fuzz the hardware: 1) fuzzing hardware like software [92, 94] 2) direct fuzzing on hardware [56] 3) fuzzing hardware like hardware [14, 23, 53]. The existing hardware fuzzers primarily focus on the RTL as their chosen level of abstraction to identify bugs.

> **Observation O1:** The chosen level of hardware abstraction and fuzzing methodologies by existing hardware fuzzers are incapable of discovering synthesis bugs, and EDA library vulnerability exploits.
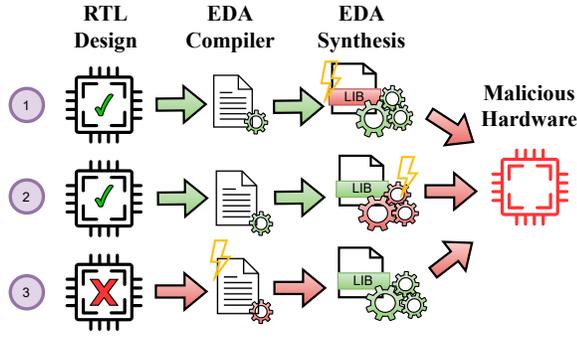
Figure 3: Overview of our Threat Models. 1) Malicious Library Mapping Exploitation 2) EDA Synthesis Tool Exploitation 3) EDA Compiler Exploitation

3 The Genesis of Synthesis Bugs: Foundations for Building Effective Fuzzers

The primary challenge in constructing an effective fuzzer for detecting synthesis bugs lies in establishing a comprehensive understanding of the threat model and addressing the inherent complexities of fuzzing to ensure efficient bug detection. For this purpose, in this section, we illustrate our threat model in Figure 3, followed by the prevailing challenges to create a hardware fuzzer to detect the bugs in Section 3.2.

3.1 Threat Model

Our Threat Model Assumptions: We assume a scenario where a malicious hardware developer exploits EDA tools or a compromised EDA library vendor leverages the optimization configurations of synthesis tools to craft a malicious hardware design.

Malicious Library Mapping: A legitimate RTL design is provided; however, during the EDA synthesis stage, a compromised library binary is provided to foster compromised logic mapping leading to malicious hardware as shown in Figure 3 ①. The EDA compiler and synthesis process appear legitimate but are tainted by the corrupted library.

EDA Synthesis Tool Exploitation: The RTL design is legitimate and free from malicious intent. However, a malicious hardware developer exploits the EDA tools during the synthesis stage to inject bugs as in Figure 3 ②. This malicious intent involves manipulating the synthesis process to introduce vulnerabilities such as deleting modules or instances, creating multiple drivers for a single signal, or driving certain pins with fixed logic constants. These actions result in unintended design behavior, leading to functional errors in the gate-level netlist. The trust placed in the EDA tool [16, 86, 98], often considered a trusted entity in the design flow, enables malicious activity by providing an avenue for hardware developers to exploit the synthesis process.

EDA Compiler Exploitation: In this threat model, an attacker crafts specific HDL behaviors that remain undetected during RTL compilation but manifest in the synthesized design. These issues

are intentionally subtle, allowing them to bypass checks during the EDA Compiler stage, which treats the RTL as legitimate, as shown in Figure 3 ③. Such flaws only surface during synthesis when the design is transformed into its gate-level representation, potentially leading to functional discrepancies or security vulnerabilities.

3.2 Prevailing Challenges

Challenge C1: Determine suitable hardware abstractions and input representations

To detect the bugs associated with the above threat models, determining the level of hardware abstraction provided to the fuzzer is critical. The fuzzing target is typically a model of the hardware design represented at the architecture level, RTL level, post-synthesis netlist level, and physical level. In addition, the hardware fuzzer should generate appropriate inputs in the format corresponding with the targeted hardware abstraction level. Thus, for efficient bug and vulnerability detection, the key challenges lie in *determining the appropriate hardware abstraction level for fuzzing and crafting input representations* that optimize the discovery of vulnerabilities within the design flow.

Challenge C2: Extract suitable coverage metrics

The second challenge is to extract suitable coverage feedback mechanisms with the selected hardware abstraction level that can detect bugs and vulnerabilities. The chosen coverage metrics should capture complex hardware behaviors at the chosen abstraction level. This helps the fuzzers to cover critical regions of the hardware design and also assists in generating efficient inputs. Hence, *the extraction of coverage metrics for feedback mechanism is crucial.*

Challenge C3: To detect bugs through a reliable reference model

The third challenge pertains to the detection of bugs, which relies upon the chosen reference model to uncover discrepancies between the expected behavior and the actual design, highlighting bugs and vulnerabilities that arise during the synthesis process. Hence, *selecting an accurate and reliable reference model is crucial for effectively identifying bugs.*

Challenge C4: Exploit the bugs and the vulnerabilities introduced through EDA tools

The final and critical challenge is to exploit the bugs and vulnerabilities or malicious modifications introduced at the synthesis stage of the hardware design flow that can evade traditional verification methods such as LEC, which are commonly used in conventional industry practices.

4 Design of SynFuzz

To address the challenges outlined in Section 3.2 and detect bugs associated with the above threat models in Section 3, we present a novel hardware fuzzer **SynFuzz** as shown in Figure 4. We fuzz the hardware at the gate-level netlist abstraction, which is mapped

to library cells by the EDA synthesis tool, representing the equivalent form of the RTL design, to detect synthesis bugs. The **SynFuzz** framework is adaptable to the conventional hardware design flow. We first introduce how **SynFuzz** performs mutations and generates an input seed format, *NetInput*, which is compatible with both the RTL and the gate-level netlist (Section 4.1). Next, we illustrate how **SynFuzz** compiles and simulates the RTL and gate-level netlist using these inputs (Section 4.2). Moving forward, we propose a coverage metric, which captures intrinsic hardware behaviors at the gate-level netlist to refine and guide the mutation of interesting seeds (Section 4.3). Lastly, we describe how **SynFuzz** cross-verifies the execution results from the RTL and gate-level netlist to identify bugs effectively (Section 4.4).
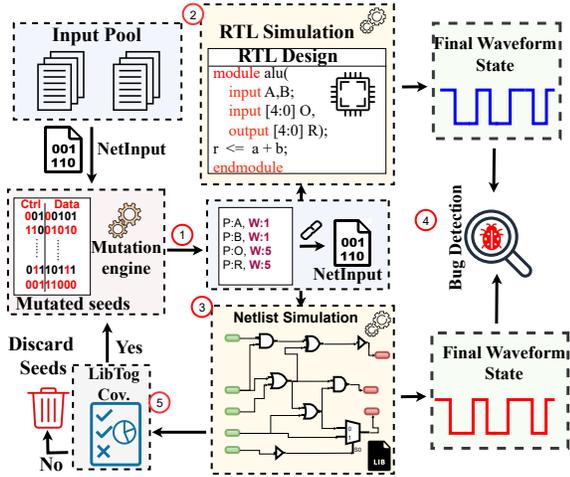

Figure 4: Overview of SynFuzz

Overview of Workflow: The overall framework of SynFuzz is shown in Figure 4. First, the randomly generated initial seeds are mutated (①), and SynFuzz runs both RTL and gate-level netlist simulation using the input (②, ③). After the completion of the simulations, SynFuzz compares the final traces of the execution results of these simulations to detect bugs (④). The seeds are further mutated using the coverage metrics to increase the exploration of uncovered hardware behaviors and improve the detection of synthesis bugs (⑤).

4.1 NetInput Seed Generation

To address Challenge C1 in Section 3.2 regarding input representation, **SynFuzz** proposes an innovative input format, *NetInput*. The gate-level netlist, which represents the synthesized equivalent of the RTL design, retains the same input ports (P) and width (W), which are concatenated to form a series of bits as shown in Figure 4. This input format is specifically designed to provide identical inputs to both the RTL and the gate-level netlist abstraction, ensuring consistency and compatibility across these two levels of hardware representation. NetInput ensures that any discrepancies or bugs identified during the fuzzing process can be directly attributed to the synthesis bugs. NetInput includes all input space that the RTL

and gate-level netlist can process during the fuzzing process. For a given NetInput, SynFuzz incorporates AFL-style mutation techniques [39]. For an efficient mutation process, the user annotates the control and data bits to the mutation engine. As shown in Figure 4 ①, two distinct mutation engines are employed: one tailored for control bits and the other for data bits. Mutating the control bits helps uncover different control paths while mutating the data bits focuses on exploring the datapath.

4.2 RTL and Netlist Simulation

SynFuzz leverages state-of-the-art EDA simulation tools to perform simulations at both the RTL and gate-level netlist abstractions. The mutated bit-vectors, which serve as input stimuli, are loaded onto the testbenches to drive simulations of the hardware designs, as depicted in Figure 4 (②, ③). For the gate-level netlist simulation, **SynFuzz** incorporates the library environment alongside the netlist. This environment is crucial for resolving and inferring the mapped instances of library cell gates, which are provided by the library vendor. These mapped instances reflect the physical implementation of the design and play a vital role in synthesis-related bugs introduced during the mapping process. **SynFuzz** focuses primarily on the intrinsic hardware behaviors at the gate-level netlist during the fuzzing process. In the following section, **SynFuzz** introduces a coverage metric designed to capture the unique characteristics of the hardware at the gate level.

4.3 Library Toggle Coverage

The hardware design represented at the RTL encompasses diverse and rich combinational and sequential circuits. The combinational circuits perform logic operations where the output depends solely on the current inputs, such as adders, multiplexers, and decoders, while sequential circuits, on the other hand, incorporate memory elements, where the output depends on both the current inputs and the stored state, as seen in flip-flops, registers, and finite state machines. During the synthesis process, these combinational and sequential components are mapped to their equivalent library gate cells provided by the standard-cell library. In the following, we demonstrate the efficacy of our proposed coverage metric in capturing the hardware characteristics associated with the mapped library gate cells.

Case Study: We illustrate with a case study using a code snippet presented in Listing 1, similar to D-link hardware authentication [25]. Firstly, we begin by describing the expected functionality and then outline the bugs. Finally, we detail how the Library Toggle coverage metrics are utilized to detect these bugs.

For Listing 1, the system should grant access by verifying the password (pwd) against the stored value in memory (mem_123), even when the superuser mode is enabled. This ensures that access is granted only to authorized users and prevents unauthorized access to the system. In addition, when the debug mode is enabled (debug), the system should check whether the input data (in_data) matches with the input data stored in the memory (mem_456). This access check prevents unauthorized or invalid inputs from being accepted under the guise of debug mode.
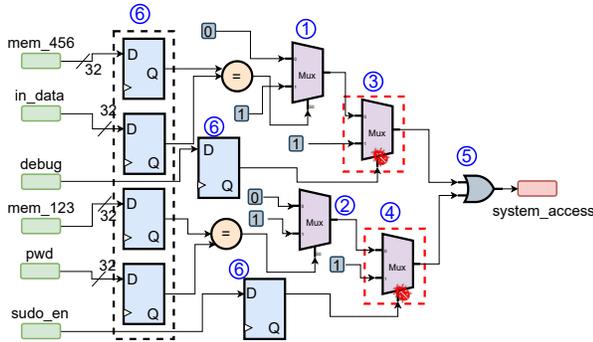
Listing 1: Access control code snippet

```
process (sudo_en, pwd, mem_123, access_level,
    debug, in_data, mem_456)
begin
if sudo_en = '1' then
access_granted <= '1';
elsif pwd = mem_123 then
access_granted <= '1';
else
access_granted <= '0';
end if;

if debug = '1' then
input_valid <= '1';
elsif in_data = mem_456 then
input_valid <= '1';
else
input_valid <= '0';
end if;
if access_granted = '1' or input_valid = '1' then
system_access <= '1';
else
system_access <= '0';
end if;
end process;
```

The EDA tools synthesize this RTL code into equivalent library gate cells, as illustrated in Figure 5. The multiplexers ① and ② handle the matching operations for the password pwd and input data in_data to determine access eligibility. Multiplexers ③ and ④ evaluate the selection of superuser mode or debug mode. The combinational logic in ⑤ controls the final system access decision, while ⑥ serves as a register to hold the input values.

Figure 5: Hardware Design for Listing 1

For the design in Figure 5, there exist two bugs, *b1* and *b2*, similar to D-Link hardware authentication. These are analogous to CVE-2021-34696 [25] (access control bypass), CVE-2021-1071 [69] (access control bypass), and CVE-2021-1088 [68] (user utilizing debug mode with insufficient access control). For the bug *b1*, as observed in Line 4 of the code snippet, if the sudo_en flag is enabled, regardless of the entered password (pwd), the password check flag (pwd_check) is set, effectively bypassing the password validation. This compromises the security of the system by allowing unauthorized access. The source of this bug is the multiplexer ③. Similarly, for the bug *b2* if the debug flag is set, then the input check (in_check) is bypassed which is associated with ④.

In ①, ②, ③, ④ all the inputs to the MUX and the corresponding select signals should be monitored for their correctness. For the

combinational logic, ⑤, it is essential to verify the intermediate computations and ensure that the logic correctly propagates the signals to grant system access. The registers ⑥ can take all possible combinations for the system access check.

For all of these scenarios, we utilize the **library toggle coverage** of the gate cells to monitor and analyze the toggling behavior of the library gate cells. This metric indicates whether a particular instance of a mapped library cell is activated or toggled during the simulation process. By tracking the toggle activity of the instances ③, ④, **SynFuzz** can detect the bugs *b1* and *b2* in the Listing 1. Furthermore, for the multiplexers ① and ②, the input data consists of constant values, which are declared using assign statements, such as assign c1 = 1'b0. These expression values play an important role in granting access when there is a password match, and the input data is valid. We use **expression coverage** to verify whether the values assigned to ① and ② are legit. **SynFuzz** leverages state-of-the-art industry standard EDA tools such as Synopsys VCS [86, 87] or Cadence Xcelium [17] to extract the library toggle coverage and expression coverage metrics. These tools have been used in the industry and academia over the decades which aligns with the traditional design verification flows. Thus, *SynFuzz's* coverage metrics aid in detecting the bugs *b1* and *b2* through the library toggle ane expression coverage metric.

4.4 Bug Detection

Upon the completion of executing both the RTL and gate-level netlist for a given *NetInput*, **SynFuzz** initiates the verification phase. Similar to traditional hardware verification, where the gate-level netlist is compared against the RTL, **SynFuzz** compares the waveform traces of the RTL and gate-level netlist to detect any discrepancies as shown in Figure 4 ④. Initially, **SynFuzz** checks whether the output register waveforms of the RTL and gate-level netlist align. If a mismatch is detected, **SynFuzz** flags the *NetInput* for the presence of a bug, which is then manually analyzed to identify its root cause. This process ensures that any inconsistencies introduced during synthesis are accurately identified and recorded for further analysis.

Challenge C5: Absence of GRM for EDA Libraries

However, note that **SynFuzz** relies on library cells during the synthesis process to identify bugs, which implies two limitations *1) Absence of GRM for EDA Library 2) Variability in Mapping Algorithms of the EDA tool and Library Characteristics*. Although the gate-level netlist is mapped to library cells to represent the equivalent RTL design, different EDA tools employ distinct algorithms for mapping the design, and library cells often vary in their logical and physical characteristics (e.g., timing, power, area). Unlike software fuzzers that rely on crashes to sanitizers and hardware fuzzers that rely on golden models, there is no GRM for EDA libraries, which introduces significant challenges in validating the instances of library cells.

Challenge C6: Variability in Mapping Algorithms and Library Characteristics

To address these challenges, in conjunction with **SynFuzz** , we introduce *DiffLib*, which extends the fuzzing process by performing differential analysis across various libraries and tools. Despite this extension, the outputs of the gate-level netlist are still compared solely with the RTL trace. By addressing these limitations, *DiffLib* complements **SynFuzz** , providing a holistic approach to detecting bugs in the synthesis process, especially those arising from tool or library-specific inconsistencies.
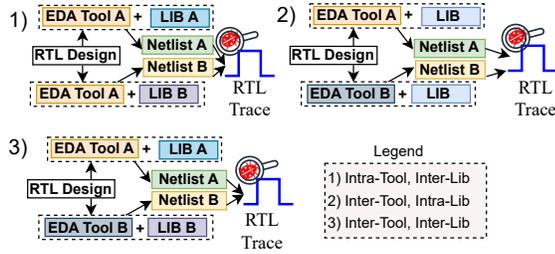

Figure 6: Configurations of DiffLib. 1) Intra-Tool, Inter-Lib 2) Inter-Tool, Intra-Lib 2) Inter-Tool, Inter-Lib

4.5 DiffLib

We find that differential fuzzing can be applied in three different ways *1) Intra-Tool, Inter-Lib 2) Inter-Tool, Intra-Lib 3) Inter-Tool, Inter-Lib* as shown in Figure 6. These approaches are designed to systematically identify inconsistencies in netlist generation by varying the tools and libraries used during the synthesis process.

Intra-Tool, Inter-Library: In this approach, netlists are generated using the same EDA synthesis tool with two different standard cell libraries, as shown in Figure 6 ①. The goal is to identify inconsistencies caused by library-specific variations, such as differences in logical functions, timing, power, or area. By keeping the synthesis tool constant, the impact of these variations on the final netlist can be isolated. For instance, two libraries may implement the same logic gate with subtle structural differences, leading to discrepancies in the synthesized netlist. This method provides insight into the synthesis tool's ability to handle diverse library properties while ensuring consistent functionality.

Inter-Tool, Intra-Library: This approach focuses on using different synthesis tools while keeping the standard cell library constant, as shown in Figure 6 ②. The aim is to detect discrepancies introduced by differences in how each synthesis tool interprets and handles the same library. Since EDA tools are developed by different vendors, they may employ unique optimization strategies, heuristics, and algorithms, leading to variances in the generated netlists even when using identical library files. This method highlights tool-specific behaviors and helps identify potential issues that may arise from discrepancies in synthesis outcomes when transitioning between tools in a design workflow.

Inter-Tool, Inter-Library: The Inter-Tool, Inter-Library approach represents the most comprehensive method, combining the variability of both synthesis tools and libraries. By testing with different

EDA tools and libraries, this approach uncovers compounded discrepancies that arise from the interaction between tool-specific behaviors and library-specific characteristics. For instance, one tool might prioritize optimizations on power, and the libraries may offer differing trade-offs for these metrics. This combined analysis provides a holistic view of the variations and inconsistencies that can occur in practical design flows, offering insights into the compounded impact of tool and library diversity.

5 Evaluation

The implementation of our proposed fuzzer, **SynFuzz**, seamlessly integrates with the conventional industry-standard IC design and verification flow, enabling efficient bug detection. The experiments were conducted on a 48-core Intel Xeon processor with 512 GB of RAM running RHEL Linux Operating System (OS). We extensively evaluate our fuzzer, **SynFuzz**, on four popular and complex open-source CPU designs: 1) the or1200 processor (OpenRISC ISA) [42], 2) the IBEX processor (RISC-V ISA) [43], 3) the PicoRV32 processor (RISC-V ISA) [43], and 4) the MIPS processor [64]. Additionally, we test it on cryptographic cores such as RSA from TrustHub [79] and Data Encryption Standard (DES) [65], a Digital Signal Processor (DSP) [50], and several IP peripherals, including UART [50], GPIO [50]. The or1200 processor is an OpenRISC based 32-bit, 5-stage pipeline processor used in academic research for multiple decades. IBEX is a 32-bit, 2-stage pipeline processor based on the RISC-V architecture. PicoRV32 is a compact 32-bit RISC-V processor designed for resource-constrained applications. The MIPS processor, a widely studied processor in academia, serves as a popular reference for educational and research purposes.

5.1 Experimental Setup

The proposed components of **SynFuzz** are designed using Python scripts and custom TCL scripts to integrate the components.

RTL and Netlist Simulation: We simulate the target hardware using a leading industry standard EDA tools, Synopsys VCS [87] and Cadence Xcelium [17]. These tools support a variety of hardware abstraction levels including RTL, gate-level, and transistor-level. To extract and analyze the coverage metrics, we deployed custom TCL and Python scripts. For deriving the netlist, we utilized state-of-the-art commercial industry standard EDA synthesis tools, Synopsys Design Compiler [86] and Cadence Genus [16], which efficiently translate the given RTL design into a gate-level representation with high performance. In addition, we also used a popular open-source EDA tool Yosys [98], which is widely adopted in the OpenROAD [2] flow for RTL synthesis and netlist generation. The above mentioned EDA tools are widely used in industry and academia for hardware research and development. They are capable of synthesizing large-scale designs and can seamlessly scale to complex CPU cores and full SoC architectures, making them suitable for evaluating real-world, high-complexity hardware systems.

These tools were provided with two open source libraries 45nm [2] and skywater130nm [2] while we use it for DiffLib. Note that, we simulate the netlist with Zero Delay Simulation (ZDS) condition. ZDS simulates the netlist without annotating any timing data. It is mainly meant for checking and validating the functionality of the design once it is translated into a gate-level netlist.

Table 1: Bugs detected in the IP designs by SynFuzz

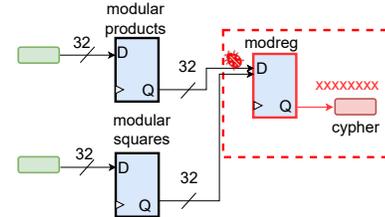| Design	Bug Description	Location	Relevant CWE(s)	New Bug ?
RSA [79]	**B1**: Multiple sources driving the modular arithmetic operations	Frontend	CWE-665, CWE-1419	✓
PicoRV32 [43]	**B2**: Incorrect memory calculation operations in Memory Interface	Memory Interface	CWE-131, CWE-119	✓
	B3: Incorrect data forwarding to co-processors	Co-Processor Interface	CWE-1422	✓
MIPS [64]	**B4**: Incorrect implementation of sum operations	ALU	CWE-682, CWE-190	✓
or1200 [42]	**B5**: Incorrect memory data retrieval from LSU	Frontend	CWE-125, CWE-119	✓
	B6: Failure of PC updation	Frontend	CWE-451	✓
	B7: Inaccurate PC updation when NPC vales in SPR changes	Program Counter	CWE-221, CWE-664	✓

Seed Generator and Mutation Engine: The seed generator, implemented using custom Python scripts, is designed to produce the initial seeds necessary for **SynFuzz**. The test suite generation is seamlessly integrated with both the seed generator and the feedback engine. Additionally, for the mutation engine, we follow the AFL mutation algorithm [39], deployed via Python scripts, and perform mutations based on the feedback provided by the feedback engine.

5.2 Bugs Reported

We now present the synthesis bug detected by **SynFuzz**. **Syn-Fuzz** found seven new bugs in our selected IP benchmarks and ten existing bugs in open-source EDA synthesis tool Yosys [98] as outlined in Table 1 and Table 2 respectively. Following this, we present the exploitation of one of these bugs via EDA tools to inject vulnerabilities in Section 5.3. Later in Section 6, we present how a malicious library exploits the EDA synthesis to foster malicious library mapping.

5.2.1 *Bugs in RSA.* SynFuzz uncovered a critical bug in the synthesized netlist of the RSA design, where discrepancies were observed between the RTL and the gate-level netlist during the generation of ciphertext. The bug, **B1**, occurs in the RSA design, where it fails to generate ciphertext for any given input text. This issue arises from multiple sources driving the modular arithmetic operations register, which results in entering an undefined state, as shown in Figure 7. The RSA performs multiple modular operations including products and squares as shown in Figure 7. As illustrated in Figure 7, we can see that both modular product and modular squares are connected to the same register modreg. Consequently, the modular operations, which are critical for the cryptographic computation and essential for generating ciphertext, become unavailable. This not only disrupts the RSA encryption process but also compromises the system's ability to perform secure and reliable cryptographic functions, leading to a denial of service (DoS) scenario for the affected hardware module. In Section 5.3, we successfully exploit this bug as a motivation to create a multiple driver in the DES crypto core through the EDA Synthesis Tools. This bug can be mapped to CWE-665 (Improper Initialization) or CWE-1419 (Incorrect Initialization of Resource), resulting in unexpected behavior or security issues.

5.2.2 *Bugs in PicoRV32.* The bug **B2**, resides within the processor's memory interface, where the address calculation for subsequent memory operations is not accurately updated. This flaw can result in disruptions to memory access, such as incorrect instruction fetching or inadvertent data retrieval from unintended memory locations, thereby compromising the reliability and integrity of the processor's operation. This bug is analogous to CWE-131 (Incorrect

Figure 7: Multiple Driver bug in the RSA

Calculation of Buffer Size) or CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), which are typically considered from a software perspective but it is being applied in a hardware context.

Bug B3: The PicoRV32 processor includes an optional interface for integrating a custom co-processor to enable hardware acceleration. However, a bug has been identified in the Pico Co-Processor Interface, where instructions meant for the co-processor are not fully fetched into the decode stage, leading to incomplete or incorrect data being forwarded to the co-processor. This bug **B3** undermines the reliability of custom instruction execution, limiting the effectiveness and compatibility of hardware acceleration through the co-processor interface. This is similar to the bug described in CWE-1422 (Exposure of Sensitive Information caused by Incorrect Data Forwarding during Transient Execution).

5.2.3 *Bugs in MIPS.* The bug **B4** resides within the Arithmetic Logic Unit (ALU) of the MIPS processor. The final output of the ALU operations fails to update when the computed result of the addition operation undergoes a change. As a result, any computational changes in the sum register do not affect the final output of the ALU, causing the ALU to erroneously retain stale data. Consequently, this will lead to incorrect arithmetic operations. This bug can be mapped to similar CWEs, such as CWE-682 (Incorrect Calculation), or it may lead to issues described in CWE-190 (Integer Overflow or Wraparound).

5.2.4 *Bugs in or1200 Processor.* The bug **B5** is in the front-end design of the or1200 processor between the Load Store Unit (LSU) and Debug Unit. The debugging unit, when performing debugging executions, requests a memory load operation. Upon receiving this request, the Load-Store Unit (LSU) retrieves the requested data from memory and makes it available to the debugging unit during the debugging process. It was detected that an incomplete data retrieval was performed from the LSU for the debugging request. This will lead to incorrect data received by the debugger leading to misinterpretation of memory states and impeding the debugging entity from diagnosing the issues in the processor. This bug can be generalized

Table 2: Detection of synthesis translation bugs in open-source Yosys EDA Tool by SynFuzz

Bug Reference#♦	Bug Description	Detected by SynFuzz ?
5105	**B8:** Incorrect optimization with Large Constant Shift	✓
5099	**B9:** Incorrect shift optimization	✓
4491	**B10:** Custom Yosys Passes Result in Faulty Synthesis and Simulation Errors	✓
4395	**B11:** Incorrect bit operation handling on empty strings	✓
4164	**B12:** Misoptimization of wide shifts bug	✓
4151	**B13:** Misoptimization of MUX Tree	✓
3895	**B14:** Inconsistency Issue with Continuous Assignment Error after FSM Optimization	✓
2969	**B15:** The optimization of for-loop doesn't produce the expected behavior	✓
2648	**B16:** The unextend option incorrectly handle sign bits	✓
1161	**B17:** Multiple Drivers changes the functionality after optimization	✓

♦ Bug Reference# refers to the Git Issue ID in the Yosys repository [99].

through relevant CWEs such as CWE-125 (Out-of-bounds Read) or CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), which can lead to unpredictable behavior or security risks.

Bug B6 arises when the instruction 95bf022d fails to execute in the synthesized netlist, resulting in an incomplete program counter (PC) update and undefined program flow. The instruction is executed, and the program counter fields are in an undefined state. This discrepancy causes disruptions in the program executions. Bug B6 can be considered analogous to CWE-451 (User Interface Misrepresentation of Critical Information), which, if not addressed, may result in incorrect execution or unintended consequences.

Bug B7 is the inaccurate update of program counter (PC) values when the values in the Next Program Counter (NPC) stored in the Special Purpose Registers (SPRs) change. The program counter is staled at the previous value when the value of NPC updates. This introduces the vulnerability of incorrect fetching of programs leading the processor to stall. Bug B7 may not have a direct mapping to a specific CWE, but it leads to issues that can be mapped to similar CWEs, such as CWE-221 (Information Loss or Omission) or CWE-664 (Improper Control of a Resource Through its Lifetime).

5.2.5 ***Bugs in Yosys EDA Tool***. Our proposed **SynFuzz** also discovered ten existing synthesis translation bugs **(B8-B17)** in the open-source EDA synthesis tool Yosys. To demonstrate the efficacy of our **SynFuzz** in detecting translation bugs introduced by EDA tools, we curated a corpus of existing bugs reported through public bug trackers [99]. Leveraging this dataset [1], we applied our proposed comprehensive fuzzing methodology—Inter-Tool, Intra-Library, Inter-Tool, and Inter-Library fuzzing strategies—as outlined in Section 4.5. Our analysis revealed that the netlists generated by commercial EDA tools such as Synopsys Design Compiler and Cadence Genus consistently adhered to the RTL-level functional specification. In contrast, the netlists produced by the open-source tool Yosys exhibited translation-induced functional bugs. The existing bugs in the open-source Yosys EDA tool detected by **SynFuzz** are detailed in Table 2, along with their corresponding bug reference numbers as reported in publicly available bug trackers.

5.3 Bug Exploitation

Based on the detection of multiple driver bugs in RSA, we intentionally replicate and introduce similar vulnerabilities into one of

[1]The RTL designs provided to **SynFuzz** were sourced from publicly available code snippets listed in the bug trackers of the Yosys EDA tool.

our benchmarks, the Data Encryption Standard (DES), by exploiting EDA tools. This approach demonstrates how EDA tools can be leveraged to introduce malicious modifications into hardware designs.

Assumption: For this purpose, we assume the role of a malicious hardware developer who exploits the state-of-the-art features of an EDA tool to transform a benign piece of hardware into a compromised or malicious component.

Multiple Driver Exploit: In the context of our findings from **Bug B1**, the creation of multiple-driver bugs involves introducing conditions where library-mapped cell gates are driven by multiple sources, leading to undefined or incorrect behavior. To successfully introduce such vulnerabilities, an attacker must first conduct a thorough analysis of the design to identify critical registers and computational components that are integral to the functionality of the IP. These components serve as prime targets for manipulation due to their significance in the operational integrity of the design.

In our specific exploit, we focused on the substitution box (S-box) of DES, which is a crucial computational block responsible for the non-linear substitution of data. The S-box is essential for ensuring the security of the encryption process, making it an ideal target for introducing a vulnerability. Once the critical components, such as the S-box, have been identified, an attacker can utilize advanced EDA tool features to deliberately create a multiple-driver scenario within the design.

This process involves deploying sophisticated EDA tool commands to establish multiple connections to the targeted components, effectively creating conflicts in the driver-load connectivity. The attacker can specify the desired drivers and loads within the tool, intentionally assigning multiple sources to drive the same gate or node. By doing so, the attacker introduces ambiguity in the design's behavior, which could lead to undefined outputs, functional failures, or exploitable vulnerabilities during operation. This exploitation highlights the risks of trusted EDA tools being misused to introduce subtle vulnerabilities, emphasizing the need for robust security, thorough validation, and secure-by-design principles in modern hardware workflows.

5.4 Coverage Analysis

In this section, we present the coverage reports for the benchmarks utilized by our fuzzer. For relatively smaller designs, such as UART, GPIO, DES, MIPS, RSA, and DSP, the fuzzer is executed for up to 8

hours. If the design achieves 100% coverage before this time, the fuzzer is terminated early. For larger designs, including or1200, IBEX, MIPS, and PicoRV32, the experiments are conducted for 12 hours to ensure comprehensive fuzzing. As mentioned in Section 4.3, we extract the library coverage for our designs.
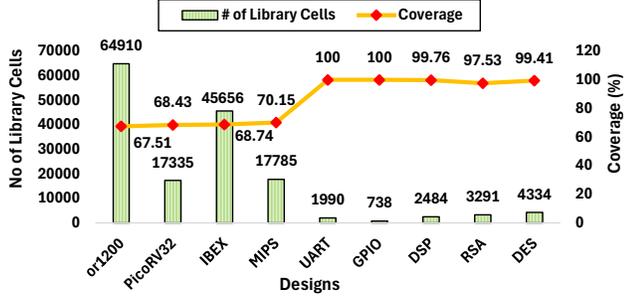

Figure 8: Coverage Results for benchmarks in SynFuzz

The coverage performance evaluation of **SynFuzz** highlights the effectiveness and adaptability of the fuzzer across a diverse range of designs. As outlined in Figure 8, or1200 is the largest design with 64190 library cells. For smaller designs such as UART, GPIO, DSP, RSA, and DES, the fuzzer achieves exceptional coverage, close to 99%. For larger designs like or1200, PicoRV32, and IBEX, with higher library cell counts, the fuzzer achieves notable coverage values of 67.51 %, 68.43 %, and 70.15%. These results are in line with some of the existing works [23, 53], and underline the proposed fuzzer's capability to handle complex architectures effectively, even within a limited fuzzing duration. On Overall, the results affirm the fuzzer's robustness and scalability, excelling in smaller designs and performing admirably on larger ones, making it a versatile tool for comprehensive synthesis bug detection.

Furthermore, we present the utilization of cell mapping as applied to our benchmarks in Figure 9. Our complex and diverse set of hardware designs effectively maps to approximately 91.70% of the cells available in the library. This high mapping percentage underscores our intent to thoroughly test the library cells under various conditions to evaluate their performance and robustness. Among the available cells, OR2X2, XNOR2X1, and XOR2X1, as well as TBUFX2, are the least utilized mapping cells. In contrast, INVX1, BUFX2, and NAND2X1 are the most highly utilized cells, reflecting their critical role and frequent occurrence in our designs.

5.5 Comparison with Logic Equivalence Checking

We also compare our **SynFuzz** with another industry standard approach - Logic Equivalence Check (LEC)[20], which is a popular technique used for the formal verification of hardware designs. During LEC, the structural and functional features of the synthesized design are compared against the RTL as a reference model after the synthesis to determine whether the two designs exhibit the same behavior. The semiconductor industry relies on commercial EDA tools such as Cadence Conformal [19] for LEC. LEC operates by mapping and comparing specific structural and functional points,
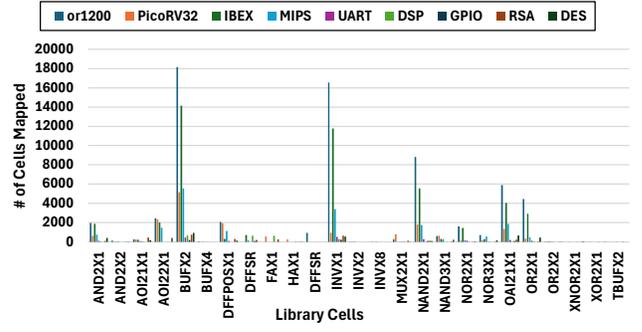

Figure 9: Cell Mapping utilized by our benchmarks

such as inputs, outputs, and intermediate signals, between the synthesized design and the RTL. These mapped comparison points are critical for identifying any discrepancies and ensuring that both designs exhibit the same behavior.

However, there are two challenges in performing these comparisons. Firstly, as modern hardware designs grow increasingly complex and larger in scale, the mapped comparison points increase exponentially; consequently, the complexity of LEC rises. This can increase the verification period to map the comparison points and verification period producing ambiguous results. Secondly, conventional LEC-based verification can perform poorly (or fail) under the following scenarios: structural optimizations introduced by EDA tools during synthesis and P&R under strict design constraints; technology translation from one standard cell library to another having different process corners or missing logic functions; untracked port/net/gate renaming introduced by EDA tools or designers; and functional/structural design transformations for IP protection like state-space transformation [77].

To ensure proper evaluations, the translated information must be accurately fed into the LEC tool, a process that demands significant human expertise and effort. The intricate nature of the mapping and comparison points, coupled with the nuances of design transformations and tool-specific optimizations, necessitates a deep understanding of both the design and the LEC tool's operation, hindering the efficiency in bug detection. The LEC fails to detect bugs found by our **SynFuzz**. Thus, in contrast to **SynFuzz**, the existing formal tools are not scalable for large designs, are prone to errors, and require significant human expertise to operate effectively.

6 CLiMA: Compromised Library Mapping Attack

In this section, we present our Compromised Library Mapping Attack (CLiMA), which exploits the optimizations of EDA synthesis tools to produce a malicious version of the hardware design. We are familiar with the fact that during the synthesis process, library binaries provided by fabrication vendors are fed to the EDA synthesis tool to map the design to library gate cells. However, an adversary or a malicious library vendor can exploit the optimizations of the EDA tool to manipulate the mapping process and create a malicious version of the hardware. To show the practicality of this attack,

we developed a malicious version of an open-source 45 nm library and used it to create a malicious variant of the or1200 CPU design. The CLiMA attack is non-detectable by traditional formal hardware verification methods such as LEC. Based on the malicious mapping process there exist two scenarios **1) Random Mapping Exploitation 2) Targeted Mapping Exploitation.** Before we present these attacks we provide some background on the library binary and the EDA tool optimizations which are being exploited.

Background: The library file (.lib) contains comprehensive information about the available gate cells, including their respective parameters such as functionality, area, power consumption, and timing characteristics. This detailed description serves as the foundation for mapping the RTL design to the appropriate standard-cell gates during the synthesis process. These are compiled by the library compilers which often emit the provided library in a binary format (.db). Since these binaries are not readable any alterations made are non traceable. A malicious vendor can manipulate the library file parameters (.lib) before it is compiled into a binary file (.db), masking the abnormalities. This tampered binary file (.db) when used by the synthesis tool, can lead to incorrect functionality or security vulnerabilities in the final design. After this, when simulating the netlist with the library environment (usually in .v), the tampered binary file causes incorrect gate mapping resulting in a functional mismatch.

To successfully execute this attack, the attacker must analyze the parameters of the library file to identify which attributes need to be tampered with to make it appear more attractive to the EDA tool. This involves understanding how the EDA tool prioritizes gate selection during synthesis based on characteristics such as functionality, timing, power, and area. By carefully manipulating these parameters, the attacker can influence the synthesis process to favor the maliciously modified gate cells, embedding vulnerabilities or altering the design's intended behavior without detection. In addition, there are design constraint commands such as maximum/minimum area, and power which influence the synthesis process. In our attack, we target for optimizing the synthesis process with minimum area constraint. In our malicious version of 45nm, we manipulate the functionality and area of certain cells to make it attractive for the EDA tool during the synthesis process. However, one can also do the same by exploiting minimum dynamic power and leakage power parameter constraints.

6.1 Random Mapping Exploitation

For implementing this attack, we selected the ALU of the or1200 processor as our target module. The ALU is a critical component responsible for performing arithmetic and logical operations, making it an ideal candidate for demonstrating the potential impact of a malicious library mapping attack. By focussing on the ALU, we demonstrate how a tampered library will lead to incorrect arithmetic operations. In our malicious version of 45nm library, the targeted cell is an AND2X1 gate where 2X1 denotes a two-input AND gate with a standard drive strength of X1. The functionality of the AND2X1 gate in the malicious version of the library is modified to perform an OR logic operation instead. Additionally, the area of these cells is intentionally reduced to be smaller than the area of the existing OR gate cells (OR2X1, OR2X2) in the library.

Table 3: Instance Mapping Before and After CLiMA

Library Cells	Original		After CLiMA	
	#	Area	#	Area
AND2X1	384	901.05	598	805.20
AND2x2	-	-	355	999.60
A0121X1	39	109.81	65	183.02
A0122X1	556	1826.51	557	1829.80
BUFX2	1662	3899.88	1627	3817.75
INVX1	2005	2822.84	2008	2827.06
NAND2X1	725	1360.97	750	1407.90
NAND3X1	105	246.38	55	129.05
NOR2X1	361	847.08	324	760.26
NOR3X1	89	250.60	55	154.86
OAI21X1	218	613.84	216	608.21
OR2X1	439	1030.11	-	-
OR2X2	2	5.63	1	2.81
Total	6585	13914	6611	13525.58

As a result, the synthesis tool prioritizes mapping the AND2X1 cells for implementing OR logic, aiming to achieve the minimum area requirements specified during the optimization process. This malicious version of the library is compiled with library compilers and passed on to the EDA tools for the synthesis process.

As outlined in Table 3, a significant reduction in the design area can be observed before and after the execution of the library mapping attack. Post-attack, there are no OR2X1 cells mapped in the design, as the malicious AND2X1 cells, modified to implement OR logic, appear more attractive to the EDA tool for meeting minimum area requirements. Specifically, 439 OR2X1 cells were replaced with AND2X1 cells due to the CLiMA attack. When simulating the netlist with the library environment, 598 cells implement AND functionality instead of OR functionality using the malicious AND2X1 cells, resulting in an erroneous change in the ALU's functionality. Hence, the root cause for this incorrect logic implementation is hard to find as this is masked in the malicious library.

Table 4: Design parameter deviations before and after the targeted mapping attack

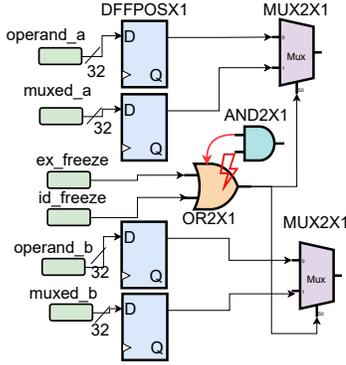|	Instances	Area (μm^2)	Power (μW)
Before	1056	2783.59	179
After	1056	+0 %	+ 0 %

6.2 Targeted Mapping Exploitation

In this attack, instead of randomly mapping cells, specific cells are deliberately targeted to create malicious behavior as shown in Figure 10. By modifying the functionality or characteristics of these chosen cells, the attacker ensures that the synthesis tool prioritizes their use in critical parts of the design, embedding vulnerabilities or altering functionality in a controlled and intentional manner. For implementing this attack, we specifically targeted the operand multiplexer module, modifying its behavior to introduce incorrect register forwarding of data. This deliberate manipulation ensures that the synthesized design mishandles data flow between registers, disrupting the forwarding to computing modules. Table 4 presents the number of cell instances and the incurred area and

Table 5: Comparison with existing HW Fuzzing Frameworks

Fuzzer	Targeted Abstraction	Input	Simulator	Coverage Metric	Target Design	Synthesis Bugs
TheHuzz [53]	RTL (HW)	Assembly	Synopsys VCS	FSM, Branch, toggle, conditional	RISC-V	✗
Processor Fuzz [21]	RTL (HW)	Assembly	Verilator	Control path register, ISA-transition	RISC-V	✗
RFUZZ [56]	FIRRTL	Series of bits	Any	Mux Toggle	Peripherals, RISC-V	✗
DifuzzRTL [49]	RTL (SW)	Assembly	Any	Register Coverage	RISC-V CPU	✗
Trippel et al [92]	RTL (SW)	Byte Sequence	Verilator	Edge Coverage	AES,HMAC,KMAC, Timer	✗
HyperFuzzer [27]	RTL (SW)	Series of bits	Verilator	High-level	SoC	✗
SoCFuzzer [48]	RTL (HW)	Byte Sequence	Xilinx ISA	Randomness, target output, input coverage	SoC	✗
HyPFuzz [23]	RTL (HW)	Assertion Cover Properties, Byte Sequence	Jasper Gold, Synopsys VCS	FSM, Branch, toggle, conditional	RISC-V	✗
SynFuzz (This work)	**Library mapped gate-level netlist**	**Series of bits**	**Cadence Xcelium**	**Library Toggle Coverage, Expression**	**CPU Designs, RSA, DES, UART, GPIO, DSP**	✓

power overhead before and after the targeted mapping attack. The results clearly demonstrate that the attack does not introduce any additional overhead, making it practically viable with no adverse impact on the performance of the design.

Figure 10: Targeted Library Mapping Attack in or1200 processor

7 Related Works

In this section, we outline the limitations of existing works in Table 5 and demonstrate how **SynFuzz** effectively detects synthesis bugs.

> **Observation O2: Comparing our work with previous approaches is not directly suitable due to inherent differences in abstraction levels and coverage metrics. The methodologies and evaluation criteria of prior works focus on specific metrics and levels of abstraction that are not aligned with the scope and objectives of our chosen level of abstraction, input format, coverage, and reference model. As such, a direct comparison would not yield meaningful insights and may lead to inaccurate and false interpretations.**

The majority of popular fuzzing frameworks, such as TheHuzz [53], DifuzzRTL [49], and ProcessorFuzz [21] are CPU design fuzzers, operating at the RTL level of abstraction and typically compare their outputs with ISA simulators, such as Spike simulator. However, the chosen level of abstraction and the input strategies used in these existing works are not efficient in uncovering synthesis bugs.

Prior works, as summarized in Table 5, have explored fuzzing hardware at different levels of abstraction. For instance, RFuzz [56] was an early effort aimed at fuzzing RTL designs by directly adopting software fuzzers and using metrics like multiplexer toggles for coverage. However, it fails to scale effectively and often misses bugs in complex designs. In contrast, Trippel *et al.* [92] proposed fuzzing hardware-like software, rather than directly applying software fuzzers to hardware designs. While this approach holds promise, the translated hardware models do not support HDL constructs or accurately account for intrinsic hardware characteristics. Moreover, the coverage metrics used in these approaches are not suitable for hardware abstractions.

Additionally, works such as [27] proposed leveraging hyperproperties— higher-level properties describing security policies by comparing system behaviors. Although useful for certain applications, these methods require significant human intervention and deep design knowledge, limiting their scalability for complex SoC designs. Later efforts [21, 23, 53] focused on fuzzing hardware at its native hardware abstraction level, as detailed in Table 5. These frameworks were designed to account for the unique characteristics of hardware systems, making them efficient at detecting bugs. However, they still rely on coverage metrics aligned with the traditional IC design flow and remain restricted to higher levels of abstraction. Based on the motivation of compiler fuzzing from software, TransFuzz [85] is proposed to fuzz the tools and compare the performance with Verismith, an open-source hardware tool fuzzer, but never focus on synthesis bugs introduced in the design.

It is important to note that a direct comparison between our work and these prior approaches is unsuitable. The abstraction levels, input strategies, and coverage evaluation metrics used in these existing works differ significantly from ours and do not apply to gate-level netlist abstraction. Notably, none of the prior frameworks focus on fuzzing at the gate-level netlist abstraction, which is the core of our methodology for identifying synthesis bugs, library bugs, and their vulnerabilities.

In contrast, our proposed **SynFuzz** 1) supports conventional hardware design and verification methods, 2) captures intrinsic hardware characteristics at gate-level netlist, 3) is efficient in detecting bugs and vulnerabilities associated with the synthesis and malicious library vendors, 4) does not require extensive design

knowledge or expertise in hardware design, and 5) is scalable large to designs with several thousands of library cells.

8 Discussion and Limitations

RTL and Netlist Accessibility: **SynFuzz** relies on access to RTL designs and EDA tools to effectively detect synthesis bugs. Verification engineers typically have access to these RTL codes during the design and verification process, enabling them to simulate and analyze the design behavior. Additionally, RTL codes can be acquired from third-party vendors at a cost, providing flexibility for organizations or engineers who require pre-designed or specialized modules for their projects. This accessibility facilitates the use of **SynFuzz** to validate both custom and third-party designs for potential synthesis vulnerabilities.

FPGA Synthesis Bugs: **SynFuzz** can be extended to fuzz FPGA-based frameworks to detect synthesis bugs in FPGA synthesis tools such as Xilinx Vivado [5] and others. In FPGA designs, device-specific library files, often referred to as bitstream libraries or technology libraries, define the mapping of high-level designs to FPGA-specific resources like Look-Up Tables (LUTs), Flip-Flops, and Block RAMs (BRAMs). These libraries play a critical role in determining the final implementation of the design on the FPGA. By targeting these device-specific libraries, **SynFuzz** can identify bugs that arise due to incorrect mapping of RTL logic to FPGA primitives. One can also perform DiffLib in the context of device libraries. Extending **SynFuzz** to FPGA frameworks allows for comprehensive validation of FPGA primitives ensuring robust and reliable hardware implementations on programmable platforms.

Fuzzing Temporal Characteristics: As mentioned in Section 5.1, all the simulations are under Zero Delay Simulation conditions ignoring all the timing intent. **SynFuzz** is not suitable for detecting any timing bugs and vulnerabilities. However, one can extend **SynFuzz** to detect timing vulnerabilities introduced by the library cells.

9 Conclusion

Hardware bugs are increasingly prevalent at different levels of abstraction during various stages of modern IC design. Existing fuzzing frameworks fall short in detecting bugs associated with synthesis and library bugs. To address this gap, we present our novel framework, **SynFuzz**, which focuses on fuzzing at the library mapped gate-level netlist. **SynFuzz** demonstrates its efficiency by identifying 7 new synthesis bugs in popular open-source designs. We presented DiffLib to extensively fuzz the EDA library in different EDA tools to uncover library vulnerabilities. By strategically exploiting EDA tool optimization settings, we presented and created a compromised library mapping attack that is undetectable by LEC. While traditional LEC methods face limitations in identifying such issues, **SynFuzz** overcomes these challenges, providing a more robust and comprehensive solution for synthesis bug detection. Compared to Conformal, a formal LEC verification tool, **SynFuzz** eliminates the need for tool-specific and design knowledge while addressing other limitations.

Responsible disclosure. The bugs have been reported to respective design maintainers.

References

[1] [n. d.]. LibFuzzer. https://www.llvm.org/docs/LibFuzzer.html Last Accessed : 6/1/2024.

[2] [n. d.]. OpenRoad. https://theopenroadproject.org/ Last Accessed : 6/1/2024.

[3] Khitam Alatoun, Bharath Shankaranarayanan, Shanmukha Murali Achyutha, and Ranga Vemuri. 2021. SoC Trust Validation Using Assertion-Based Security Monitors. In *International Symposium on Quality Electronic Design (ISQED)*.

[4] Aldec. [n. d.]. Riviera-PRO: Advanced Verification Platform. https://www.aldec.com/en/products/functional_verification/riviera-pro Last Accessed : 6/1/2024.

[5] AMD. [n. d.]. AMD Vivado Design Suite. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado/vivado-buy.html Last accessed: 09/09/2024.

[6] N. Nalla Anandakumar, M. Sazadur Rahman, Mridha Md Mashahedur Rahman, Rasheed Kibria, Upoma Das, Farimah Farahmandi, Fahim Rahman, and Mark M. Tehranipoor. 2022. Rethinking Watermark: Providing Proof of IP Ownership in Modern SoCs. Cryptology ePrint Archive, Paper 2022/092.

[7] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[8] Nitay Artenstein. 2017. BROADPWN: REMOTELY COMPROMISING ANDROID AND iOS VIA A BUG IN BROADCOM'S Wi-Fi CHIPSETS. In *BlackHat USA*.

[9] A.Yeh. [n. d.]. Trends in the Global IC Design Service Market. https://www.digitimes.com/news/a20120313RS400.html&chid=2 Last Accessed : 6/1/2024.

[10] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2018. SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018), 97–122.

[11] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2020. NNgSAT: Neural Network guided SAT Attack on Logic Locked Complex Structures. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[12] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2021. From Cryptography to Logic Locking: A Survey on the Architecture Evolution of Secure Scan Chains. *IEEE Access* (2021).

[13] Swarup Bhunia and Mark Tehranipoor. 2018. *Hardware Security: A Hands-on Learning Approach* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors. *arXiv preprint arXiv:2402.03704* (2024).

[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[16] Cadence. [n. d.]. Cadence Webpage. https://www.cadence.com/en_US/home.html Last Accessed : 6/1/2024.

[17] Cadence. [n. d.]. Fastest Simulator to Achieve Verification Closure for IP and SoC Designs. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html Last Accessed : 6/1/2024.

[18] Cadence. [n. d.]. JasperGold Formal Verification Platform. https://www.cadence.com/enUS/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html Last accessed: 11/18/2023.

[19] Cadence Inc. [n. d.]. Conformal Smart LEC. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/logic-equivalence-checking/conformal-smart-lec.html Last accessed: 01/19/2025.

[20] Cadence Inc. [n. d.]. ogic Equivalence Checking. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/logic-equivalence-checking.html Last accessed: 01/19/2025.

[21] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. Taylor, M. Egele, and A. Joshi. 2023. ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.

[22] Rajat Subhra Chakraborty and Swarup Bhunia. 2009. HARPOON: An obfuscation-based SoC design methodology for hardware protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (2009), 1493–1502.

[23] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. HyPFuzz: Formal-Assisted Processor Fuzzing. 1361–1378.

[24] James C. Chen, Hsin Rau, Cheng-Ju Sun, Hung-Wen Stzeng, and Chia-Hsun Chen. 2009. Workflow design and management for IC supply chain. In *International Conference on Networking, Sensing and Control*. 697–701.

[25] Cisco. [n. d.]. CVE-2021-34696 in Cisco Routers. https://sec.cloudapps.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-asr900acl-

UeEyCxkv Last Accessed : 6/1/2024.

[26] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. *Model Checking and the State Explosion Problem.* Springer, 1–30.

[27] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium.* 51–65.

[28] Aritra Dasgupta, Sudipta Paria, and Swarup Bhunia. 2025. HIPR: Hardware IP Protection through Low-Overhead Fine-Grain Redaction. Cryptology ePrint Archive, Paper 2025/553. https://eprint.iacr.org/2025/553

[29] NIST National Vulnerability Database. [n. d.]. https://nvd.nist.gov/ Last Accessed : 6/1/2024.

[30] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer-Verlag, 337–340.

[31] Stephanie Drzevitzky. 2010. Proof-Carrying Hardware: Runtime Formal Verification for Secure Dynamic Reconfiguration. In *2010 International Conference on Field Programmable Logic and Applications.* 255–258.

[32] Carson Dunbar and Gang Qu. 2014. Designing Trusted Embedded Systems from Finite State Machines. *ACM Trans. Embed. Comput. Syst.* 13 (2014), 153:1–153:20. https://api.semanticscholar.org/CorpusID:14568179

[33] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. 2017. Trojan localization using symbolic algebra. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC).* 591–597.

[34] Nusrat Farzana, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi. 2019. SoC Security Verification using Property Checking. In *IEEE International Test Conference (ITC).*

[35] Tian Feng, Haojie Pei, Zhou Jin, and Xiao Wu. 2022. A survey and perspective on electronic design automation tools for ensuring SoC security. In *International SoC Design Conference (ISOCC).*

[36] Domenic Forte, Swarup Bhunia, and Mark M. Tehranipoor. 2017. *Hardware Protection through Obfuscation.*

[37] Antonis Geralis. [n. d.]. LibFuzzer. https://github.com/planetis-m/libfuzzer Last accessed: 11/18/2023.

[38] Prokash Ghosh, V. N. Dwaraka Mai, Aditya Chopra, and Baljinder Sood. 2023. Self-Checking Performance Verification Methodology for Complex SoCs. In *International Symposium on Quality Electronic Design (ISQED).*

[39] Google. [n. d.]. Americal Fuzzy Loop. https://github.com/google/AFL Last Accessed : 6/1/2024.

[40] Google. [n. d.]. OSS-Fuzz. https://google.github.io/oss-fuzz/ Last Accessed : 6/1/2024.

[41] Daniel Grosse, Ulrich Kuhne, and Rolf Drechsler. 2005. HW/SW Co-Verification of a RISC CPU using Bounded Model Checking. In *2005 Sixth International Workshop on Microprocessor Test and Verification.* 133–137.

[42] OpenRISC Working Group. [n. d.]. OpenRISC. https://openrisc.io/ Last accessed: 11/18/2023.

[43] RISC-V Working Group. [n. d.]. RISC-V. https://riscv.org/ Last accessed: 11/18/2023.

[44] Xiaolong Guo, Raj Gautam Dutta, Jiaji He, and Yier Jin. 2017. PCH framework for IP runtime security verification. In *2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST).* 79–84.

[45] Xiaolong Guo, Raj Gautam Dutta, and Yier Jin. 2017. Eliminating the Hardware-Software Boundary: A Proof-Carrying Approach for Trust Evaluation on Computer Systems. *IEEE Transactions on Information Forensics and Security* 12, 2 (2017), 405–417.

[46] Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin. 2017. Automatic Code Converter Enhanced PCH Framework for SoC Trust Verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 12 (2017), 3390–3400.

[47] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. 2015. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. *SIGPLAN Not.* 50, 4 (Mar 2015), 517–529.

[48] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. SoCFuzzer: SoC Vulnerability Detection using Cost Function enabled Fuzz Testing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[49] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *IEEE Symposium on Security and Privacy (SP).*

[50] IEE. [n. d.]. CAD for Assurance. https://github.com/CommonEvaluationPlatform/CEP/tree/master Last accessed: 09/09/2024.

[51] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security.*

[52] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective Processor Verification with Logic Fuzzer Enhanced Co-Simulation. In *IEEE/ACM International Symposium on Microarchitecture.*

[53] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *USENIX Security Symposium (USENIX Security).*

[54] Kyungsu Kang, Sangho Park, Byeongwook Bae, Jungyun Choi, SungGil Lee, Byunghoon Lee, and Jong-Bae Lee. 2019. Seamless SoC Verification Using Virtual Platforms: An Industrial Case Study. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[55] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P'19).*

[56] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).*

[57] T Li, H Zou, Luo D, and Qu W. 2021. Symbolic simulation enhanced coverage-directed fuzz testing of RTL design. In *IEEE International Symposium on Circuits and Systems (ISCAS).*

[58] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: A Language for Hardware-Level Security Policy Enforcement. In *International Conference on Architectural Support for Programming Languages and Operating Systems.*

[59] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flow. *SIGPLAN Not.* 46, 6 (June 2011), 109–120.

[60] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium.*

[61] Yu Liu, Yier Jin, Aria Nosratinia, and Yiorgos Makris. 2017. Silicon Demonstration of Hardware Trojan Design and Detection in Wireless Cryptographic ICs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 4 (2017), 1506–1519.

[62] Xingyu Meng, Shamik Kundu, Arun K. Kanuparthi, and Kanad Basu. 2022. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2022), 466–477.

[63] Microsoft. [n. d.]. Microsoft Security Risk Detection. https://www.microsoft.com/en-us/research/project/project-springfield/ Last Accessed : 6/1/2024.

[64] MIPS Technologies. [n. d.]. MIPS. https://mips.com/ Last Accessed : 6/1/2024.

[65] MIT-LL. [n. d.]. Common Evalution Platform. https://github.com/CommonEvaluationPlatform/CEP/tree/master Last accessed: 09/09/2024.

[66] Pratheema Mohandoss and Archana Rengaraj. 2018. Pre-Silicon DFT Verification on SoC Slim Model. In *International Workshop on Microprocessor and SOC Test and Verification (MTV).*

[67] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. 2020. Hyper-Fuzzing for SoC Security Validation. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD).*

[68] NVIDIA. [n. d.]. CVE-2021-1088 in NVIDIA GPU and Tegra Hardware. https://nvidia.custhelp.com/app/answers/detail/a_id/5263 Last Accessed : 6/1/2024.

[69] NVIDIA Corportation. [n. d.]. CVE-2021-1071 in NVIDIA Hardware. https://nvidia.custhelp.com/app/answers/detail/a_id/5147 Last Accessed : 6/1/2024.

[70] University of California Berkley. [n. d.]. ABC. https://people.eecs.berkeley.edu/~alanmi/abc/ Last Accessed : 6/1/2024.

[71] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. 2023. DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection. arXiv:cs.CR/2308.06932

[72] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. 2024. DiSPEL: A Framework for SoC Security Policy Synthesis and Distributed Enforcement. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST).* 271–281. https://doi.org/10.1109/HOST55342.2024.10545407

[73] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. 2024. Navigating SoC Security Landscape on LLM-Guided Paths. In *Proceedings of the Great Lakes Symposium on VLSI 2024 (GLSVLSI '24).* Association for Computing Machinery, New York, NY, USA, 252–257. https://doi.org/10.1145/3649476.3660393

[74] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. 2024. SPELL: An End-to-End Tool Flow for LLM-Guided Secure SoC Design for Embedded Systems. *IEEE Embedded Systems Letters* 16, 4 (2024), 365–368. https://doi.org/10.1109/LES.2024.3447691

[75] Sudipta Paria, Pravin Gaikwad, Aritra Dasgupta, and Swarup Bhunia. 2024. LATENT: Leveraging Automated Test Pattern Generation for Hardware Trojan Detection. In *2024 IEEE 33rd Asian Test Symposium (ATS).* 1–6. https://doi.org/10.1109/ATS64447.2024.10915238

[76] Md Moshiur Rahman, Rasheed Almawzan, Aritra Dasgupta, Sudipta Paria, and Swarup Bhunia. 2024. United We Protect: Protecting IP Confidentiality with Integrated Transformation and Redaction. In *2024 IEEE Physical Assurance and Inspection of Electronics (PAINE).* 1–7. https://doi.org/10.1109/PAINE62042.2024.

10792848

[77] Md Moshiur Rahman and Swarup Bhunia. 2023. Practical Implementation of Robust State-Space Obfuscation for Hardware IP Protection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023).

[78] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. 2013. Security analysis of integrated circuit camouflaging. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013).

[79] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. 2013. On design vulnerability analysis and trust benchmarks development. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 471–474.

[80] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. 2024. The Emergence of Hardware Fuzzing: A Critical Review of its Significance. arXiv:cs.CR/2403.12812 https://arxiv.org/abs/2403.12812

[81] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security*.

[82] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*.

[83] Siemens. [n. d.]. Questa Advanced Verification. https://eda.sw.siemens.com/en-US/ic/questa/ Last accessed: 11/18/2023.

[84] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security Symposium*.

[85] Flavien Solt and Kaveh Razavi. 2025. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. In *USENIX Security*. Paper=https://comsec.ethz.ch/wp-content/files/mirtl_sec25.pdfURL=https://comsec.ethz.ch/mirtl

[86] Synopsys. [n. d.]. Synopsys Webpage. https://www.synopsys.com/ Last accessed: 11/18/2023.

[87] Synopsys. [n. d.]. VCS: The Industry's highest Performance Simulation Solutions. https://www.synopsys.com/verification/simulation/vcs.html Last accessed: 11/18/2023.

[88] Shibo Tang, Xingxin Wang, Yifei Gao, and Wei Hu. 2022. Accelerating SoC Security Verification and Vulnerability Detection Through Symbolic Execution. In *International SoC Design Conference (ISOCC)*.

[89] TechInsights. [n. d.]. Apple iPhone 15 Pro Teardown. https://www.techinsights.com/blog/apple-iphone-15-pro-teardown Last Accessed : 6/1/2024.

[90] Mark Tehranipoor, Kimia Zamiri Azar, Navid Asadizanjani, Fahim Rahman, Hadi Mardani Kamali, and Farimah Farahmandi. 2024. *SoC Security Verification Using Fuzz, Penetration, and AI Testing*. 183–229.

[91] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *International Symposium on Computer Architecture (ISCA)*.

[92] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *USENIX Security Symposium (USENIX Security)*.

[93] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. 2008. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits* 43, 1 (2008), 29–41.

[94] Verilator. [n. d.]. Welcome to Verilator. https://www.veripool.org/verilator/ Last accessed: 11/18/2023.

[95] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bogdan, and Shahin Nazarian. 2018. Accelerating Coverage Directed Test Generation for Functional Verification: A Neural Network-Based Framework. In *Great Lakes Symposium on VLSI*.

[96] Bruce Wile, John Goss, and Wolfgang Roesner. 2005. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann Publishers Inc.

[97] Rafal Wojtczuk. [n. d.]. Xen Security Advisory 7 (CVE-2012-0217) - PV privilege escalation. https://lists.xen.org/archives/html/xen-announce/2012-06/msg00001.html Last accessed: 11/18/2023.

[98] Yosys. [n. d.]. SymbiYosys Documentation. https://symbiyosys.readthedocs.io/en/latest/ Last accessed: 11/18/2023.

[99] Yosys. [n. d.]. SymbiYosys Documentation. https://github.com/YosysHQ/yosys/issues Last accessed: 09/09/2024.

[100] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.